NASA Contractor Report 178384

The Computational Structural Mechanics Testbed Architecture: Volume I - The Language

(NASA-CR-178384) THE COMPUTATIONAL STRUCTURAL MECHANICS TESTBED ARCHITECTURE. VOLUME 1: THE LANGUAGE (Lockheed Missiles and Space Co.) 95 p CSCL 20K

N89-14472

Unclas

G3/39 0165073

Carlos A. Felippa

Lockheed Missiles and Space Company, Inc.
Palo Alto, California

Contract NAS1-18444

December 1988

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Preface

The first three volumes of this five-volume set present a language called CLAMP, an acronym for **Command Language for Applied Mechanics Processors**. As the name suggests, CLAMP is designed to control the flow of execution of Processors written for NICE, the **Network of Interactive Computational Elements**, an integrated software system developed at Lockheed's Applied Mechanics Laboratory.

The syntax of CLAMP is largely based upon that of a 1969 command language called NIL (NOSTRA Input Language). The language is written in the form of free-field source command records. These records may reside on ordinary text files, be stored as global database text elements, or be directly typed at your terminal. These source commands are read and processed by an interpreter called CLIP, the **Command Language Interface Program**. The output of CLIP does not have meaning *per se*. The Processor that calls CLIP is responsible for translating the decoded commands into specific actions.

The ancestor of CLIP, LODREC, was patterned after the input languages of ATLAS and SAIL, two structural analysis codes that evolved at Boeing in the late 1960s. More modern language capabilities, notably command procedures and macrosymbols, have been strongly influenced by the Unix™ operating system and the C programming language, as popularized by Kernighan, Plauger and Ritchie in their textbooks. The Unix "shell/kernel" concept, in fact, permeates the architecture of the NICE system, of which CLIP is a key component.

NIL and its original interpreter LODREC, which now constitutes the "kernel" of CLIP, has been put to extensive field testing for over a decade. In fact NIL has been the input language used by all application programs developed by the author since 1969 to 1979. (NIL also drives the relational data manager RIM developed by Boeing for NASA LaRC.) During this period many features of varying degree of complexity were tried and about half of them discarded or replaced after extensive experimentation. CLAMP represents a significant enhancement of NIL, particularly as regards to directive processing, interface with database management facilities, and interprocessor control. The current version is therefore believed to be powerful, efficient, and easy to use, and well suited to interactive work.

The present Manual is a greatly expanded version of the original March 1980 version, revised on April 1981. Because of its length, the material has been divided into five Volumes, which cater to different user levels.

Volume I (NASA CR-178384) presents the basic elements of the CLAMP language and is intended for all users. Volume II (NASA CR-178385), which covers CLIP directives, is

intended for intermediate and advanced users. Volume III (NASA CR-178386) deals with the CLIP-Processor interface and related topics, and is meant only for Processor developers. Volume IV (NASA CR-178387) describes the Global Access Library (GAL) and is intended for all users. Volume V (NASA CR-178388) describes the low-level input/output (I/O) routines.

All volumes are primarily organized as reference documents. Except for modest attempts here and there (*e.g.* §3.1 in Volume I and Appendix C in Volume III), the presentation style is not tutorial.

Contents

1
Introduction

§1.1 WHAT IS A COMMAND LANGUAGE?

Readers not previously exposed to interactive software may find it difficult to grasp the difference between a programming language such as FORTRAN or Basic, and a command language. The key differences are summarized below.

1. Programming languages are used to construct executable software elements such as FORTRAN subroutines, Pascal procedures or Ada packages. On the other hand, command languages are used to *guide* the high-level execution of such software elements. Put it in another way: programming languages are used to specify data processing and management functions, while command languages are primarily used for high-level control functions.

2. Programming languages are generally *compiled* into object code prior to linking and execution. Command languages are *interpreted* at run time by control software.

3. Command languages specify *actions*, which in most cases are carried out immediately before the next command is read. The do-it-now mode facilitates conversational operation of programs by interactive users, because these users can enter commands in response to the observed effect from the previous command. This "improvisational", continuous-feedback style cannot be achieved with conventional programming languages.

Some batch-oriented users may be surprised to learn that they have been using a command language for some time! The so-called "control cards" in batch operating systems are nothing more than statements of an *operating system command language* through which the user directs the overall job execution of operating system routines. In this case, the software being controlled is the computer operating system.

A *problem-oriented command language* is one through which the user controls the execution flow of *application programs*. In this case, the thing being controlled is the application software itself. The qualifier *problem-oriented* means that the English-like command syntax reflects the application. For example, a command appropriate for a finite-element analysis code might be

```
PRINT ELEMENTS 5 TO 24
```

which is easily memorized. This example clearly shows that command languages tend to be of higher level than programming languages, because the details of how the print display is accomplished are concealed and the user simply perceives the results of the `PRINT` request.

Or to put it more succinctly: in a command language the task of specifying *how* is less important than specifying what.

REMARK 1.1

In the computer science literature, conventional programming languages such as FORTRAN, Pascal or Ada are sometimes called *procedural*, whereas higher level command languages are called

nonprocedural. These adjectives try to convey the idea of how versus what, but are misleading in the sense that one may certainly write command language "procedures". (A good part of Volume II is devoted to this topic).

REMARK 1.2

The term *object-oriented programming* is currently in vogue to describe software-development methodologies that emphasize thinking in terms of *objects* whose actual representation in the computer is irrelevant to the user. For example, a finite element is an object; a subroutine that prints finite element data is an object, and so on. Command languages combine functional abstraction (the verb `PRINT` in the example) with object references (the names that follow the verb).

REMARK 1.3

Note that the example command was printed in typewriter font. This convention is used throughout this Volume set: it means the *actual* command as typed by the user. This is different from a command specification, which is done in terms of a *metalanguage* described in §10. The metalanguage specification combines typewriter font for literal items with italics font for variable items.

§1.2 WHAT IS CLAMP?

CLAMP is an acronym for **Command Language for Applied Mechanics Processors**. The name conveys the origin and intended application.

In general terms, CLAMP was created to simplify high-level, interactive operation of application programs and integrated networks of such programs. It offers program developers ways and means for building *problem-oriented languages* tailored to achieve specific goals. The language is not tied, however, to any specific application: it is *generic*.

More specifically, CLAMP was originally designed and implemented to support the NICE system, which has been under development at Lockheed's Applied Mechanics Laboratory since 1980. But, as noted above, the scope of CLAMP is not limited to NICE support.

The CLAMP language may be logically viewed (see Remark 1.4 below) as a stream of free-field *command records* read from *command sources*. Command sources may be actual files or virtual files (messages). The source commands are interpreted by a "filter" utility called CLIP, which stands for **Command Language Interface Program**. The main function of CLIP is to produce *object records* for consumption by its user program. (In this regard, see Remark 1.5 below.)

Most (but not all) command records are *devoid of meaning* at the CLIP level. That is, CLIP does not care what the command is for. This is analogous to a data management system, which does not care about the meaning of the data structures it manages, or a compiler, which does not care about the purpose of the code it translates. Going back to the example of §1.1, CLIP interprets the command

```
PRINT ELEMENTS 5 TO 24
```

as a sequence of five items: `PRINT`, `ELEMENTS`, `5`, `TO` and `24`. But CLIP does not understand about finite elements, element numbers, and similar problem-related things.

The assignation of meaning transmutes object command records into *statements*. Statements are the basic building blocks of a *problem-oriented language*. The language *drives* the application program through the statements. A command-driven input environment is ideally suited to interactive work, whether carried out in conversational or spectator run mode. (For precise definition of these "run mode" terms, see the Glossary provided in Appendix A.)

REMARK 1.4

CLAMP represents a significant enhancement of the NOSTRA Input Language (NIL), which was developed to support the NOSTRA program during the period 1971-1972. Historically curious users may read Appendix B.

REMARK 1.5

Not *all* commands are devoid of meaning at the CLIP level: §2 introduces *directives*, which are commands executed directly by CLIP. Volume II is entirely devoted to directive description.

§1.3 THREE COMMAND VIEWS

In modern database management systems three views of the stored data are distinguished: *physical*, *logical* and *conceptual*. A similar three-tier structure can be distinguished for command languages:

1. *Physical view*: data lines. Commands as physical records of characters. This level relates to the way you get these characters into the program.

2. *Logical view*: command records. Commands as item sequences. This level is also called the *syntactical or grammatical level*: you learn a set of rules for writing formally correct commands, e.g., that items are separated by blanks. This level is concerned with form and not with meaning.

3. *Conceptual view*: statements. Commands as action agents. This level is also called the *semantic level*: you are concerned with what a syntactically correct command will do for you.

At the physical level CLIP is simply a free-field data line reader. At the logical level CLIP is a *lexical analyzer* that parses those data lines into tokens or *items*. CLIP enters the conceptual level only for directives. For ordinary commands the conceptual level is left to the processor executive.

§1.4 MANUAL ORGANIZATION OUTLINE

This document has been written to fulfill two main objectives.

1. To serve as a CLIP User's Manual for developers of application programs (*e.g.*, NICE processors) that use CLAMP as source input language.

2. To serve as a general Reference Manual for CLAMP language syntax and command descriptions.

The material covered in Sections 2 through 10 applies to both objectives, although the last three sections deal primarily with advanced features.

This Manual is *not* a tutorial document.

2

CLIP

§2.1 CLIP OPERATION

The command language interpreter CLIP interacts with three operational elements: user, running processor, and global data manager. These three terms are defined as follows:

User. The person (or entity) that reaps the benefits of the processor activity. In interactive conversational mode, a human user is in direct two-way communication with the processor. In other situations, such as batch runs, the communication is indirect and occurs through a prepared command file.

Running Processor. The software element that produces results for the user. (If the running processor is a NICE-conforming processor, the noun is capitalized: Processor.) The processor activity is controlled by the command language.

Global Data Manager. The software element through which the global database is accessed. The global database is a catalogued collection of data produced by the running processor or other communicating processors. It can also store command language procedures and help documentation for the interactive user. The global data manager for the NICE system is called GAL.

During processor execution, CLIP can operate in three modes: user command, user directive, or message. The last mode has two variants known as self-message and mailbox. These operation modes are summarily described below.

User Command Mode

The standard operating mode of CLIP is the *user-command mode*, sometimes called the *command mode* for short. Commands are directly supplied by the user, retrieved from ordinary card-image files, or extracted from the global database, and submitted, on request, to the running processor. This mode can be diagrammed as

$$\left.\begin{array}{r}\text{User}\\ \text{Card-image file}\\ \text{Database entity}\end{array}\right\} \rightarrow \text{Command text} \rightarrow \text{CLIP} \rightarrow \text{Processor}$$

User Directive Mode

In *user directive mode*, special commands called *directives*, which are supplied by the user, read from card-image files, or retrieved from the global database, are processed directly by CLIP. The processor is "out of the loop". This mode can be diagrammed as

$$\left.\begin{array}{r}\text{User}\\ \text{Card-image file}\\ \text{Database entity}\end{array}\right\} \rightarrow \text{Directive text} \rightarrow \text{CLIP} \; [\rightarrow \text{Database}]$$

where the bracketed term is meant to indicate that processing of the directive may possibly affect the database state.

Directives are identified by a leading keyword prefixed by an asterisk. Transition from processor-command mode to directive mode is automatic. Once the directive is processed, CLIP returns to processor-command mode unless the directive is a command-procedure definition. In this exceptional case, CLIP stays in the directive mode until the end of a procedure is detected (procedure definition is the only instance of a multi-command directive).

Directives are used to dynamically change run-environment parameters, to process advanced language constructs such as macrosymbols and command procedures, to implement branching and cycling, and to request services of general usefulness.

All CLIP directives are available from *any* processor that uses it.

Self-Message Mode

Message mode means that the processor "talks" to CLIP. There are two variants of message mode: self-message and mailbox. In self-message mode, the processor supplies a directive or stream of directives, possibly intermixed with ordinary commands, to CLIP for immediate processing. This mode may be diagrammed as

Processor → *Message text* → CLIP [→ Database]

In this mode the user is "out of the loop." Transfer to message mode occurs when the processor calls a message-sender entry point. The processor-CLIP ensemble stays in this mode until an explicit or implicit end-of-message signal is acknowledged by CLIP.

Mailbox Mode

This is an advanced variation of the message mode in which the processor supplies a command procedure to CLIP for downstream consumption by other processors. CLIP effectively acts as a *mailbox* through which the command procedure is stored in the global database. This mode can be diagrammed as

Processor → CLIP → Database → CLIP → Processor

Again the user is "out of the loop." As above, transfer from the processor-command mode to the message mode is initiated by the running processor calling a message-sending entry point. The key difference from the previous case is that the message is not immediately "opened" by CLIP, but simply saved for use by another processor. The mailbox mode forms the basis for synchronizing the execution of NICE mini-networks for coupled-system dynamic analysis or optimization applications.

§2.2 INTRA- AND INTERPROCESSOR CONTROL

The conventional use of a command language such as CLAMP is to guide the flow of execution within the running processor: do this, do that. This is called *intra-processor control.* The key feature is that the processor does not stop executing.

In certain advanced network-operation modes a running processor may directly or indirectly initiate the execution of other processors, put itself to sleep (hybernation) and resume execution (wakeup). These operations pertain to a higher plane known as *inter-process control.*

In addition to its more mundane capabilities, certain versions of CLIP provide interprocessor control services. These services include process initiation, suspension, task synchronization through status checking, and wakeup. These services are provided through *ad hoc* directives.

The implementation of interprocessor control is highly machine-dependent, because CLIP has then to talk directly to the operating system. Remnants of the dependency percolate to the processor-developer and processor user. On some archaic operating systems, such as Univac's Exec-1100 and CDC's Scope, these operations are either impossible or severely restricted.

On the last account, effective application of interprocessor control services is a fairly advanced and specialized topic and thus it does not pertain to the elementary exposition level of this Volume. The subject is dealt with in the "SuperCLIP" chapter of Volume II.

§2.3 CONFIGURATION OF CLIP

Internal Structure

The present CLIP consists of the following components.

CLIP Shell. The implementation of a closed interface between the three internal components of CLIP (kernel, directive and database-interface subsystems), and the external environment. (For the definition of "closed interface," see Appendix A.)

CLIP Kernel. Largely based on LODREC (cf. Appendix B) The workhorse module of CLIP. Performs tasks pertaining to the decoding and parsing of command language text.

Directive Subsystem. Controls activities undertaken while in directive mode. For example, definition and retrieval of command procedures.

Global Database Interface Subsystem. A module that handles communication with the global database manager GAL for activities that require transactions with the global database.

SuperCLIP. A module that handles activities that pertain to interprocessor control. This module exists only under certain operating systems.

CLIP Versions

The CLIP master source code (MSC) embeds all conceivable instances of CLIP, including machine-dependent code for various computer systems such as VAX/VMS, VAX/Ultrix, SUN/UNIX and CRAY/UNICOS. Virtually all of the code is full FORTRAN 77, with some assembly-language sections for VAX/VMS.

The single MSC file also contains CLIP versions of varying functionality. These versions are delimited by the MAX distribution keys listed in Table 2.1.

When the MSC file is read through the preprocessor MAX to extract compilable code, specifying all the above distribution keys results in a version of CLIP with its full functionality. If no keys are specified, a bare-bones CLIP results; this stripped version (MicroCLIP) is not much more than a fancy free-field reader. Specifying several (but not all) keys yields versions of intermediate functionality.

Table 2.1. **CLIP MSC Distribution Keys**

Key	*Purpose*
COMPRO	Delimits the command procedure facility described in §5 of Volume II. This is part of the directive subsystem.
EZGAL	Delimits the global data manager interface described in §7 of Volume II.
MACRO	Delimits the macrosymbol facility described in §4 of Volume II.
SUPERMAN	Delimits the interprocessor control facility known as Super-CLIP, which is described in §9 of Volume II. This is only presently available under VAX/VMS.
WORKPOOL	Delimits the local data manager interface described in §8 of Volume II.

3
Commands

§3.1 WHAT DO COMMANDS LOOK LIKE?

This section covers the general aspects of the command language CLAMP. Before launching into technical details, however, let us begin with an overview of what CLAMP commands are supposed to look like. If you are already an experienced user of command languages, you may want to go over this section quickly, just skimming over it to absorb terminology.

The following description covers the so-called *standard CLAMP format*. This is a small but important subset of the total number of command formats that CLIP can process. Reasons for using this particular format are offered in §3.2.

One-Item Commands

The simplest type of command has only one item, which is usually an action verb. Examples:

```
RUN
STOP
```

An item such as `RUN` or `STOP` is called the *command verb*. The verb indicates what the command does. These commands are very easy to remember and quick to type, so they are recommended for highly interactive programs as long as they are workable.

Two more advanced but important applications of one-item commands are: to enter and exit *processor subsystems*, and as components of multilevel commands. The last function is briefly covered in Remark 5.1.

Abbreviations

One-item commands are so much easier to type that sometimes the command language designer "cheats" a little bit. Consider

```
PRINT TABLE_OF_CONTENTS
```

This is a two-item command in which `TABLE_OF_CONTENTS` is a either a parameter of the `PRINT` command, or a verb modifier, as explained below. If this command happens to be heavily used by interactive users, the command designer may introduce an *abbreviation* such as

```
TOC
```

There is no visible verb here; `PRINT` is implied. As a general rule the abbreviation technique should be sparingly used, as it can get out of hand. It is generally better to let users decide upon their own "custom" abbreviations.

Parameterized Commands

One-item commands are convenient but limited in function. Most useful commands are *parameterized* in one way or another. For example:

```
TYPE INPUT.DAT
```

is a parameterized command that may request that the contents of a text file be displayed on the user's terminal. Here TYPE is the command verb while INPUT.DAT is a parameter. If you look up the description of the TYPE command on the Users Manual or Processor Help File, you ought to see a "generic" description such as

> "The command
>
> TYPE *Filename*
>
> displays the contents of an existing card-image file named *Filename* on the user's terminal."

This description style is typical of CLAMP commands. The key point is that *Filename* is a parameter; hence the use of italics; capitalization of the first letter conventionally indicates that a character string is expected (precise rules to this effect are given in §10.2). When the user enters a TYPE command, he or she writes the name of the specific file to be displayed.

Parameter Lists

Parameters need not be single items. Some commands take *parameter lists*. A list is a sequence of items separated by commas. Example:

```
DELETE  4,17,23,31
```

The four integers: 4,17,23 and 31 form a parameter list for the DELETE command.

Is the order of list items relevant? It may or may not be: this depends on the command function and its implementation. Consultation of the directives manual or Processor Help File is recommended to determine exact usage.

Assignment Commands

A more general form of a parameterized command is one in which a parameter, or parameter list, is equated to another parameter, or to a parameter list. Examples:

```
SET SPEED OF SOUND = 6125.3
DEFINE NODELIST = 1,2,3,5,8,13,21
COPY 1,2 = 3,6,ABSTRACT
```

This command form is typically used to "assign" or "instantiate", in some sense, objects named in the right parameter list to the objects named on the left. Think of the general form

Verb *Destination* ← *Source*

or, if you are mathematically minded,

Verb *Lefthandside* ← *Righthandside*

in which the *Verb* clarifies the operation. These are called *assignment commands.* Typical "assignment" verbs: ASSIGN, ATTACH, BIND, CONNECT, COPY, DEFINE, GET, MAP, MOVE, PUT, RENAME, SET, TRANSFER. Note the similarity with assignment statements in conventional programming languages, for example

```
A = B + X        (FORTRAN)
a := b + x;      (Pascal)
```

Verb Modifiers

Sometime a command verb is actually two words; the second one, called a *modifier*, makes the first one more explicit:

```
SET UNIT PRINT = 6
```

SET UNIT is a compound command verb; UNIT is the modifier. PRINT is the left (receiver) parameter and 6 is the right (source) parameter.

Qualifiers

So far we have talked about commands whose items are mandatory; they must not be omitted and must appear in the order shown. For example, leaving out UNIT in SET UNIT PRINT = 6 produces

```
SET PRINT = 6
```

whose meaning is quite different.

How can we take care of *options*? An elegant way, though far from the only one, is through *command qualifiers.* A qualifier is a word preceded by a special prefix, which in CLAMP is usually the slash. Example:

```
OPEN /NEW INPUTFIL
```

may be a command that opens a *new* file called INPUTFIL. The blank before the / is often optional, but it never hurts.

The key feature of a qualifier is that it is *optional*, which means that there is always a *default interpretation.* If you just say:

```
OPEN INPUTFIL
```

this must be a legal command for opening INPUTFIL. (A common default interpretation, by the way, is open an old file if it exists or else create a new one.)

In most cases, the position of a qualifier does not matter as long as it comes after the verb. Thus

```
OPEN INPUTFIL /NEW
```

will also work. This indifference to position is an asset in conversational interactive work, as the need for a qualifier often comes as an afterthought, after one has typed much of the non-default part.

The slash is the usual default qualifier in CLAMP commands, but it may be changed to another special character through the use of the SET character directive.

Parameterized Qualifiers

Sometimes qualifiers are followed by a parameter or parameter list, to which they are connected by an equals sign. Here is an example: the command

```
OPEN INPUTFILE /NEW /LIMIT=45000
```

specifies the ***capacity*** of file `INPUTFIL`, which is the maximum size to which the file may expand after creation; this is necessary on some archaic operating systems. Just saying `/LIMIT` would not work; the computer has to be told "how big it can get." It is perfectly acceptable to have a default capacity; therefore, `LIMIT` is a qualifier and not a parameter.

§3.2 STANDARD CLAMP FORMAT

Are the commands illustrated in §3.1 the only forms accepted by CLIP? Far from it. CLIP can process virtually *any* command format you care to think about.

Thus, if you happen to be a command designer with Germanic tastes and would like to put the verb at the end, so be it. CLIP will pass to your program the parsed command and *you* search for the verb. Similarly, you can have three parameter lists or transpose the meaning of = in an assignment command, or replace the   by the keyword TO.

Even if you object to CLIP item-parsing rules there is an ultimate solution: you can ask for the "virgin" text, just as the user typed it, and write your own parser. (Volume III explains how to get the command text.)

Why, then, have we talked about a standard format? There are three good reasons that put together form a compelling case.

1. *Additional Support.* CLIP provides a comprehensive set of utilities for helping you, as a *processor developer*, process a standard-format command. For example, you can ask for a list of all qualifiers and you will receive it. But if your command syntax is nonstandard you will have to build such utilities yourself.
2. *Interface Consistency.* Components of large program networks such as NICE are written by different persons. Adopting a standardized command format avoids surprising users when they move from one processor to another.
3. *Agreement with CLIP Directives.* As noted previously, directives are special commands processed by CLIP. Their format agrees with that described in §3.1. Thus, interface consistency is further enhanced.

§3.3 COMMAND SOURCES

Source Files

After reading or skimming §3.1, you should have by now an idea of what CLAMP commands look like. But where do they come from? Well, CLIP *normally* takes them from a card-image file. Reasonably enough, this file is called the *command source file.*

But wait, you say. Suppose I am happily typing commands at my terminal: what file are you talking about? Well, there is always a file involved, albeit invisible. Most operating systems communicate with the outside world of peripheral devices through actual files, and a terminal is a peripheral device. Thus, the lines you type become in fact records of the *system input file*. Since CLIP is inside the computer, it can access your keyboard input only by reading that file.

REMARK 3.1

On most systems, CLIP accesses the system input file through the READ (unit,'(A)') statement of FORTRAN 77.

REMARK 3.2

The system input file has system-dependent names. On the Univac it is called READ$; on CDC it is INPUT; and on VAX/VMS it's SYS$INPUT. But the name is unimportant. To CLIP this is called the *root* command source because of the reasons offered in §4.2, and its internal name is either **`$root`** or **`$term`**.

REMARK 3.3

Why did we say "normally takes them"? There is the world of messages to worry about. One-line messages are not implemented as actual disk files, but virtual ones.

Changing Sources

When CLIP starts up, its input is normally taken from the system input file. The command source file can be identified with other files or source of input. CLIP can in fact change the source input for

(A) Reading commands from a "script" or procedure file;

(B) Reading commands from a global database entity; or

(C) Receiving commands sent by the processor while in the message mode.

A change in the command source file to account for cases (A) and (B) occurs in response to special directives. In case (C), the switch occurs in response to a processor reference to the message-receiver entry point in CLIP. More details on multisource input are provided in §4.2.

§3.4 THE COMMAND STREAM

The sequence of images received by CLIP from one or several source files is called the *command source stream*, or *command stream* for short. The command stream is a *logical* concept; it reflects the way CLIP kernel "sees" its input:

command1
command2
command3
...

regardless of physical source. Perhaps *command1* comes from the terminal, *command2* from a script file, *command3* from a message. From the receiving end, it does not really matter.

Following §4, which deals with physical aspects, §5 and §6 focus on the basic organization of the command source stream. These two sections explain how the basic command components, called *items*, mesh together to form command records. Rules to this effect define the command record syntax. §7 through §9 go deeper and cover rules concerning individual items.

Basic Characteristics

Some important characteristics of the command stream are listed next.

1. The command stream is partitioned into logical blocks of information called *command records.*

2. On first cut, command records may be categorized into *ordinary commands* and *directives.* Ordinary commands are processed by CLIP for eventual use by the running processor and become object command records on output. Directives are for internal consumption by CLIP, although indirect effects may be felt by the processor. (A more refined classification takes into account the *command sender*: user or processor.)

3. Each call to the "get next command" entry point by the running processor loads one and only one *ordinary* command record. This rule is *never* violated.

4. Any directives interspersed between two ordinary commands, say C_1 and C_2, are processed by CLIP when the processor calls for C_2. The number of intervening directives is irrelevant. It can be one or one thousand. A procedure-definition block (introduced by a `PROCEDURE` directive and terminated by an `END` directive) counts as one directive.

4

Lines of Input

§4.1 DATA LINES

Each ***physical record*** of the command source stream is said to be a *data line* or simply *line*. Character positions within a line are sometimes called *columns*, a terminology holdover from the good old days of punched-card input.

How are data lines created? If commands are entered directly from an online keyboard device, lines are composed "on the spot" by the interactive user and sent to CLIP when the carriage-return key is pressed. If input is to be read from an existing card-image file, data lines are prepared in advance (for example, with a text editor) and then CLIP is told the file name. In the case of punched-card input for a batch run, each card is effectively one data line.

Data Field

The ***data field*** is the "active" portion of a data line. It is the portion examined by CLIP for command items. The extent of the data field is governed by a simple rule: the data field extends from column 1 through the end-of-line mark, or column 80, whichever occurs first. (See Remark 4.1 below for a generalization).

End-of-line marks may be explicit or implicit. This topic is taken up in detail in §5.

Sentinels

The first character of each data line may have a special significance, in which case it is called a *sentinel character*. The following two sentinels are presently recognized:

.	(period)	If followed by a blank, it flags a comment line (which may be echoprinted, but is otherwise ignored).
@	(at)	Causes an end-of-source condition detectable by the data line reader.

REMARK 4.1

The standard 80-column line width limit may be widened or contracted through the directive **SET WIDTH** directive described in §53 of Volume II.

REMARK 4.2

The period is the default comment line sentinel. It may be changed to another special character through the **SET CHARACTER** directive discussed in §53 of Volume II.

REMARK 4.3

In batch mode, the image of each data line is immediately printed upon being read by CLIP. This echo display can be suppressed, however, with the **SET ECHO** directive discussed in Volume II. In the interactive mode, the echo display is normally turned off, but can be turned on with the **SET ECHO** directive.

§4.2 MULTIPLE SOURCE INPUT

Multiple source input occurs frequently in nontrivial operation of CLIP. Therefore, it's important that prospective users get a general idea of how this capability works so that they can make informed decisions about data preparation for complex problems.

The Command Source Stack

CLIP keeps track of multisource input through a *command source stack* (CSS), the top of which points to the active input source. The CSS operation is best described by working through an example.

Assume that a NICE Processor is activated and that the first thing it does is to call a CLIP entry point and state: "give me the first command". When CLIP receives this request, it tries to read the first data line, since the first command is presumably within the data line. But where is the first line?

CLIP normally assumes that the first data line arrives from the *root command source file*. The term "root" relates to the appearance of this source at the *base* of the command source stack. Within CLIP this file is conventionally known as Source number 0 (zero) and has the internal name `$root` for batch or spectator-interactive work or `$term` for conversational-interactive work. In the examples below we assume the latter. (As noted in §3.3, for interactive operation, this source is the user's terminal, whereas for batch it is the system-defined card-image input file.) So at runstart the CSS contains only one entry:

Stack level	*Source no.*	*Source name*	
0	0	`$term`	(1)

After some commands have been processed, an ADD directive tells CLIP to open the existing "script" file `INPUT.DAT`, and begin reading commands from it. CLIP connects this to an internal FORTRAN unit, let's say 32, and places this source on top of the CSS:

Stack level	*Source no.*	*Source name*	
1	32	`INPUT.DAT`	
0	0	`$term`	(2)

Data lines stored in unit 32 are sequentially accessed through FORTRAN reads, and commands contained in those lines processed. Suppose a call to procedure `ITERATE` is encountered.

As noted in §2, command procedures may reside on ordinary (editable) card-image files or on data-library files readable through the global database manager GAL. Assume it's the latter. Data library files are internally identified by Logical Device Indices (LDI), which range from 1 through 30. The data library that contains the procedure is `PROCLIB.GAL`, and its LDI is 3. Once CLIP begins reading data from the command procedure, the CSS looks like:

Stack level	*Source no.*	*Source name*	
2	3	ITERATE	
1	32	INPUT.DAT	
0	0	$term	(3)

Two points should be noted: (1) CLIP stores the negative of the LDI in the CSS to distinguish a procedural source from a non-procedural source such as unit 32, and (2) the name recorded in the CSS is **ITERATE**, not **PROCLIB.GAL**. (Had the procedure been resident on an ordinary file, the name of the file must be **ITERATE**, so in such a case there is no dichotomy.)

Now suppose that procedure **ITERATE** contains an **ADD** directive that opens another existing script file **MAKEITGO.DAT**, connects it to unit 33, and directs subsequent reads to it. The CSS now contains four entries:

Stack level	*Source no.*	*Source name*	
3	33	MAKEITGO.DAT	
2	-3	ITERATE	
1	32	INPUT.DAT	
0	0	$term	(4)

Eventually the end-of-file (EOF) on unit 33 is reached. CLIP then closes unit 33 and "pops" the stack – that is, CLIP discards the top level of the last-in-first-out (LIFO) queue. We are back to the three-level configuration (3), and CLIP continues reading the command procedure. When the end of the procedure is detected, or a return-from-procedure taken, the stack is popped again, to the two-level configuration (2), and reading continues from unit 32. On end-of-file on unit 32, the CSS reverts to its one-level original configuration (1) and input is back to the root command source. Should an end-of-file be detected at this point (for example, an interactive user enters an @ in column 1), CLIP notices that the stack is exhausted, and calls the run-termination routine **ENDRUN** described in Volume III.

REMARK 4.4

The command source stack contains additional information not described here. The complete "packet" of information is called a CSS *frame*. Frames may be displayed through the **SHOW CSS** directive covered in §54 of Volume II.

REMARK 4.5

The first data line seen by CLIP may actually come from the processor rather than unit 0 if the processor starts up execution by sending a message. This is fairly common in NICE Processors, where the message may be used to set "startup" options. On the VAX it is also possible that the first line comes from the processor invocation as a foreign DCL command.

REMARK 4.6

Nothing prohibits the root input source from appearing more than once in the CSS. For example

Stack level	*Source no.*	*Source name*
5	4	PRINTALL
4	0	$term
3	33	MAKEITGO.DAT
2	-3	ITERATE
1	32	INPUT.DAT
0	0	$term

This sample configuration may appear in interactive mode if a command read from procedure **ITERATE** asks for user's feedback, and in response the user has called upon procedure **PRINTALL**. Repetition of non-procedural source units such as 32 should not occur, however, because such files are read sequentially, and a multiple-positioning conflict would occur. On the other hand, repetition of procedural sources is not only feasible but common in practice; for example, a procedure may call itself.

REMARK 4.7

The CSS concept permits uniform implementation of multiline messages as internal ADD files, and of procedural recursion (a command procedure may call itself directly or indirectly).

Summarizing

The use of an input stack allows uniform treatment of heterogeneous command sources. These sources may be procedural or non-procedural. There are few restrictions on the order in which sources can appear. The stack depth is limited to 6 levels but in practice 2 or 3 is rarely exceeded. An end-of-file or return-from-procedure always acts as a *return* to the previous input source as long as at least one remains in the stack. If none remains, an end-of-file acts as a normal run stop.

§4.3 RESERVED SOURCE UNITS

The use of units 32 and 33 in the example of §4.2 is not accidental. FORTRAN logical unit numbers 30 through 40 are in fact reserved for usage by CLIP.

Unit 30 is a reserved unit used for dynamic connection to non-command sources such as help files. Unit 31 is a reserved scratch file where CLIP saves command-procedure header and state tables. The running processor should not tamper with these two units.

Units 32 through 40 are available for connection to non-procedural card-image files named in **ADD** directives. These units do not have to be pre-assigned, however, because they are automatically connected and disconnected by CLIP as needed. Selection of this particular sets of units minimizes the chance of clashing with files under control of the global data manager GAL.

REMARK 4.8

On some operating systems, unit number constraints may force a different block to be reserved.

5

Command Records

§5.1 COMMAND STRUCTURE

In this section, we rise from the physical view of data lines to a higher plane - the logical level. In §3.4 it was stated that the source stream is logically subdivided into *command records*. A command record is a block of symbolic information processed by CLIP as a whole. In the case of an ordinary command, CLIP does not return control to the running processor until the entire command record is interpreted.

It was also noted in §3.4 that a command record can be an ordinary command or a directive. Directives are treated exhaustively in Volume II. Consequently, in this and following sections of this Volume, attention is directed to ordinary commands unless otherwise noted.

An ordinary command record is a sequence of items terminated by an explicit or implicit end-of-record. These records are interpreted by CLIP and presented to the running processor, which is supposed to carry out the action(s) specified by the command.

A command record, while subject to the size limitations stated in §5.2, may extend over any number of data lines. Conversely, several command records may be written on the same line if space allows.

REMARK 5.1

There is generally a one-to-one correspondence between processor actions and command records, but sometimes several command records may be used to compose one statement. The most common instance of this is *detailed prompting*. For example, take up again the sample statement of §1.1:

```
PRINT ELEMENTS 5 TO 24
```

If this is a common command, the processor may allow it to be "broken up" to help a beginner user:

Enter command:	`PRINT`
Print what:	`ELEMENTS`
Range:	`5 TO 24`

The text on the left of the : is prompting text written to the screen, and the text on the right is the user's response. In this example, the `PRINT` statement is made up by *three* command records.

§5.2 ITEMS

Command records are formed by components called *items*. An item is a string of printable characters appearing on the data field. Items are delimited by blanks, commas, special delimiters, or data field boundaries, and may be written anywhere inside the data field, *i.e.* in *free-field* form. (A complete description of item delimiters is provided in §6).

The interpretation of a command by CLIP is essentially a process of *item evaluation*. Evaluation means looking at what the user has typed and figuring out the appropriate interpretation in terms of primitive data types such as integer, floating-point, or character string.

An *expression* is an item or a combination of items that eventually evaluates to a *single* value.

Item Categorization

Items can be categorized into three types:

1. *Data Item*: an item whose value can be determined directly from the characters entered by the user. For example:

```
ABC      432      1.765E+6
```

2. *Special Item*: work-saving constructions such as

```
1:101:10      35@2.5
```

which specify numeric list generation and item repetition, respectively, or marks used for special purposes such as delimitation of command records.

3. *Symbolic Item*: an item or expression that has to go through a string-replacement process for evaluation. In some cases, the replacement process may be quite complicated and involve multilevel nesting. The final product is one or more data items. CLIP handles two types of symbolic items: *procedure arguments*, and *macrosymbols*. As the use of both types depends heavily on the notion of directives, they are not described in this Volume.

REMARK 5.2

Previous versions of CLIP (and its ancestor LODREC) incorporated since 1972 a third symbolic item type: the *register*, which has disappeared from the present version. Its function (primarily that of controlling loops in command procedures) has been taken over by a special form of macrosymbol described in §4 of Volume II.

Data Items

Data items may be numeric or nonnumeric. The latter are called *character strings* and often function as command keywords at the processor level. Numeric items may be integer or floating-point constants. These types are extensively studied in §7.

Further material on the classification and interpretation of special items and symbolic items is provided in §8-9 of this Volume, and in Volume II.

Size Limitations

The present version of CLIP allows up to 512 data items to appear on an ordinary command record. If this limit is exceeded, an informative diagnostic is issued and the excess items are discarded. The item count does not include symbolic or special items *per se*, but does include data items generated as a result of the processing of symbolic or special items.

There are two other size limitations that pertain to character count:

1. The total length of a command record, *excluding* in-line comments or interspersed comment lines, may not exceed 2400 characters. Obviously this limit is only relevant when you have lots of continuation lines (at least 30). If this limit is exceeded, CLIP first tries to squeeze out multiple blanks from all data lines read so far. If this device fails an informative diagnostic is printed and trailing continuation lines are ignored.

2. The sum of the character lengths of all character-string data items may not exceed 480. If this limit is exceeded, an informative diagnostic is given and excess characters are discarded.

§5.3 RECORD MARKS

Termination

A command record can be terminated explicitly or implicitly.

Explicit termination. The following special character sequences are interpreted as *explicit* end-of-record marks:

;	blank-semicolon
.	blank-period-blank

The semicolon mark is used to separate short records written on the *same* data line. This saves space on prepared command files (scripts or procedures) that are to be archived, but offers no special advantages in conversational work.

An isolated period indicates that the next command record begins on another line: text following this *end-of-line* terminator is ignored. This feature may be exploited to insert inline comments in nonvolatile command files such as scripts or command procedures that are to be archived for some time.

Implicit termination. A command record is implicitly terminated when the right data field boundary (column 80 or carriage return mark) is reached without a continuation mark being encountered.

REMARK 5.3

Implicit termination is by far the most common in conversational interactive work, where pressing the return key is equivalent to ending the data field. In conversational mode all command source input is usually volatile, i.e. disappears upon processing by CLIP; thus explicit termination marks serve no useful purpose and just add keystrokes.

REMARK 5.4

If the isolated period is in column 1, the whole line is treated as a comment line (see §4.1).

REMARK 5.5

Both the record separator and the end-of-line mark may be changed to another character via the **SET CHARACTER** directive.

Continuation

A long command record may be extended over the next data line by writing one of the continuation marks:

++	blank-plus-plus
--	minus-minus

before the right data field boundary is reached.

The double-plus mark must be preceded by a blank to be recognized, and is ignored if inside an apostrophe string (§7.6) or a quote string (§7.7). This mark is always an item delimiter, *i.e.* items cannot be continued into the next line.

The double-minus mark may be used as a *hyphenation* mark to continue a long item into the next line, and is recognized even inside an apostrophe string or a quote string. This property is occasionally useful for things such as very long textstrings, *e.g.*

```
'If I have all the eloquence of men or of angels, --
but speak without love, I am simply a gong boom--
ing or a cymbal clashing'
```

A double-minus is *not* treated as a hyphenation mark inside an apostrophe or quote string that is *closed* on the same line. For example:

```
TITLE = 'This is the way it was -- and will be'
```

The double minus sign is not a hyphenator here because there is a matching apostrophe in the same line.

There is no *a priori* limit on the number of continuation lines; however, the limitations on number of items, total record size and character-string-sum size stated in §5.2 should be kept in mind when writing very long command records.

REMARK 5.6

Both continuation marks may be changed to another character pair (or be disabled) via the **SET CHARACTER** directive explained in Volume II.

Examples

To illustrate the most important rules stated above, consider the following command record:

```
LOAD INPUT CASES 6 TO 9 LEVEL 32.4 .   one record
```

This command record contains eight data items. Items 1, 2, 3, 5 and 7 are character strings. Items 4 and 6 are fixed-point constants (integers). Item 8 is a floating-point constant. The isolated-period end-of-line is *not* counted as a data item. Next, consider

```
LOAD INPUT ; CASES 6 TO 9 ; LEVEL 32.4 .   three records
```

We now have *three* command records written on the same data line. Note that semicolon separators must be preceded by at least one blank; otherwise they would be treated as part of the preceding item. Finally, consider

```
LOAD INPUT        ++     First line
CASES 6 TO 9      ++     Second line
LEVEL 32.4        .      Third and last line
```

This represents *one* command record that extends over three data lines. (Double-pluses may be replaced by double-minuses with identical effect in this example.) Observe that anything appearing after a continuation mark or an end-of-line mark is treated as comment text.

§5.4 EMPTY LINES

An empty line is one that contains only zero or more blanks, or one or more blanks followed by an inline comment. Unless told otherwise (see Remark 5.6), CLIP *ignores* all empty lines, just as it ignores comment lines.

The most visible effect of this feature is in conversational mode. Suppose that you start up a processor and get the following prompt on the screen:

Enter something:

If you respond to this request with a carriage return, you will see the same prompt come up instantly: CLIP is still waiting! If you type one thousand carriage returns in a row, you will get a thousand prompts but nothing else will happen.

The same thing will happen if you space over and type a carriage return, or just enter an inline comment:

Enter something: `. I am not ready!`
Enter something:

Empty continuation lines are also ignored. Example:

```
BEGIN LIST = 1, 2, 3, --
    . this is an empty continuation line
4, 5, 6, --

7, 8, 9
```

This input sequence has the same effect as

```
BEGIN LIST = 1, 2, 3, --
4, 5, 6, --
7, 8, 9
```

Note that the continuation mark does not have to appear explicitly in empty continuation lines.

REMARK 5.7

The "ignore empty lines" behavior is the normal one. CLIP may be told to pay attention to empty lines through the `SET MODE` directive discussed in Volume II. The non-default interpretation is useful for batch-oriented processors that use a multilevel command language and key on an empty line for detecting the end of an input block. This interpretation is not recommended, however, for interactive processors because accidentally typing blank lines is a common occurrence.

6
Characters

§6.1 THE CHARACTERS YOU TYPE

Commands are character streams, so CLIP is intimately acquainted with the world of characters. This Section focuses on the characters you type to form commands, and the special attributes that some of these characters enjoy.

CLIP is not tied to any particular character set, but all of its implementations so far have been on ASCII machines. (ASCII stands for American Standard Code for Information Interchange.) The only serious competitor to ASCII is presently EBCDIC, which is used on IBM mainframes. So for the sake of specificity the text below refers to characters that you will normally find on the so-called ASCII keyboards.

The ASCII Character Set

ASCII is a 7-bit integer code, which spans 0 through 127 inclusive. It has 94 visible characters, which are internally coded 33 to 126, inclusive. There is also the blank or space, which is coded 32. Visible characters and the blank are called *display*, *visible* or *printable* characters. Each of the printable symbols should be on your keyboard.

The remaining ASCII characters, coded 0 to 31 and 127, are *control* or *nonprintable* characters. They are used to send signals to the operating system, to format your screen displays, etc. With a few important exceptions such as escape, delete and return, these characters do not have dedicated keyboard keys, and must be created by control sequences. For example, on a VAX running under VMS, <control-Y> creates a process-interrupt character.

Printable Characters

The set of 95 printable characters include three families:

Letters: A-Z and a-z. As explained in more detail in §6.3 and §7.6, upper and lower case letters are *usually* equivalent because CLIP internally converts the latter to the former unless the letters are "protected" with enclosing apostrophes. The choice between lower and upper case is therefore largely a matter of personal style.

Numbers: 0-9. No ambiguity here.

Non-alphanumerics. The remaining printable symbols are

" # $ % & @ * + - = , . : ; ? ! ( ) < > [ ] { } \ / | ^ ~   ‘ ’

plus the blank.

Special Characters

Some of the non-alphanumeric characters shown above assume special significance in CLAMP, and so they are the primary subject of the following sections. These characters are covered in alphabetic order, as per the table:

Character	*See*
Angle brackets	§6.2
Apostrophes	§6.3
Arithmetic Operators	§6.4
Asterisks	§6.5
At Signs	§6.6
Blanks and Commas	§6.7
Colons	§6.8
Dollar Signs	§6.9
Equals Signs	§6.10
Parentheses	§6.11
Percent Signs	§6.12
Periods	§6.13
Quotes	§6.14
Semicolons	§6.15
Slashes	§6.16
Square Brackets	§6.17

REMARK 6.1

The following non-alphanumeric characters do not have special significance in the present implementation of the language:

& ~ ! ? ‘ # \ { }

and remain available for user-defined chores. Paraphrasing Luigi Pirandello, we might call them characters in search of a purpose. One possible use is substituting one of the above characters for a special character in present use through the **SET CHARACTER** directive.

§6.2 ANGLE BRACKETS

Balancing left-right angle brackets function as delimiter pairs for macrosymbol references (§4 of Volume II). For example:

```
<SOLVE(mtx1; mtx2; result; LOADS=1.2,1.45,1.52,1.6)>
```

Angle brackets *should not be used for any other purpose unless inside apostrophe or quote strings*. Otherwise, CLIP will complain about undefined macrosymbols.

§6.3 APOSTROPHES

Apostrophes are character-string delimiters of higher precedence than any other except hyphenation marks in the case discussed in §5.3. More precisely: with the exception of the double-minus hyphenation mark not followed by an apostrophe in the same line, any character that appears within apostrophe marks, including blanks, commas, equal signs, and the like, is considered part of the string. Example:

```
'1, 2, 3, 4, 5'
```

This is a 13-character string and not a list of integer items. An apostrophe can be represented as part of the string by repeating it as in FORTRAN 77; for example:

```
'Don't get me wrong'
```

represents the string `Don't get me wrong`.

A common use of apostrophes is the specification of long textstrings for labelling print or plot output.

Apostrophe delimiters enjoy another special property on computers where the character set distinguishes between upper and lower case letters (this is true of all modern computers except CDC Cybers). CLIP automatically *converts all lower-case letters to upper case, unless such letters are enclosed in apostrophes.* This strategy aims at protecting lower-case letters for things such as print titles or plot legends, as in

```
PLOT XLABEL = 'Circular Sampling Frequency omega*h'
```

while simplifying keyword decoding by the processor (because keywords need be tested only against upper-case strings).

§6.4 ARITHMETIC OPERATORS

The following six characters:

*	asterisk
^	caret or hat
-	minus
%	percent sign
+	plus
/	slash

are used as operators in the specification of the arithmetic expressions discussed in §7.4.

The asterisk, percent sign and slash have other special uses discussed in §6.5, §6.12 and §6.16, respectively.

§6.5 ASTERISKS

The ubiquitous asterisk was used as a multipurpose special character in old versions of CLIP and even more heavily in its ancestor LODREC. In the present version, however, asterisks have only two special uses:

1. Prefix of directive verb (Volume II). Example:

```
*SHOW ARGUMENTS
```

2. Multiplication operator in the arithmetic expressions treated in §7.4. For example,

```
SET LIMIT = (<pi>*(2^0.5))
```

   The expression above, by the way, evaluates to $\pi\sqrt{2}$.

Aside from these two cases, asterisks are now treated as an ordinary nonnumeric character. For example:

```
A*B, *, 6*, **, *24
```

This is a list of five character strings: `A*B`, `*`, `6*`, `**` and `*24`. In older (pre-1982) CLIP versions, the last four would have been treated as special items.

§6.6 AT SIGNS

The at-sign character @ has two special uses:

1. Item repeater when prefixed by an integer. Example:

   ```
   4@(1/3)
   ```

   means that item (1/3) is to be repeated four times.

2. End-of-command-source sentinel, as described in §4.1.

Aside from these two cases, the at-sign is treated as an ordinary nonnumeric character.

REMARK 6.2

In pre-1983 CLIP versions, the asterisk served as an item repeater, in a construction that mimicked the value repetition in FORTRAN DATA statements. This invited confusion when arithmetic expressions were introduced, as further discussed in Remark 7.13.

REMARK 6.3

The second use of @ has historical roots: the extensive use of LODREC on the Univac 1100 from 1971 to 1980. On that machine, an at-sign on column 1 indicates a "control statement" and terminates a data deck. The custom has survived the Univac name (the machine is now called Sperry).

§6.7 BLANKS AND COMMAS

Blanks are the conventional "white space" *item delimiters*. In fact, CLIP ignores any blank not comprised between higher-precedence delimiters such as apostrophes or quotes.

Commas delimit items just as blanks do, but serve an additional function: *specifying item lists*. Example:

```
1 2 3 4 5
1,2,3,4,5
```

In the second form, integers 1 through 5 are logically connected to form a five-integer list. This association does not exist in the first form. The distinction has implications as regards use of the list-loading entry points described in Volume III.

REMARK 6.4

The precise meaning of commas is as follows. Each data item processed by CLIP is stored in a *Decoded Item Table* that remembers its type (integer, floating, or character), its value, and two characters called ***prefix*** and *separator*. The five items: **1,2,3,4,5** are stored in that table as follows:

Type	*Pre*	*Value*	*Sep*
Integer		1	,
Integer		2	,
Integer		3	,
Integer		4	,
Integer		5	

Thus, commas are "remembered" as separators. Suppose that the processor then asks for this particular list. CLIP delivers items to the processor until a separator other than a comma is found.

REMARK 6.5

The presence of commas does not affect individual item retrieval. For example, the processor may call for the third item in **1,2,3,4,5** and the value **3** is returned regardless of the presence or absence of commas. The underlying philosophy is: provide higher level functions such as list retrieval, but do not block processor developers that want to do more primitive things.

Consecutive commas, or commas separated only by blanks, generate interspersed zero items; see §9.2 for additional details.

§6.8 COLONS

Colon delimiters are used to separate components of numeric list generators described in §8.3. Thus

```
1:15:2
```

generates the integer list

```
1, 3, 5, 7, 9, 11, 13, 15
```

Colons are not delimiters within character strings; for example

```
PROC:FORPRC.MSC
```

(a VAX/VMS file name) is a single character string.

REMARK 6.6

A character string prefixed by a colon is interpreted as a *label* in *command procedure* constructions that involve nonsequential command execution, such as branching and looping. A label can occur only as an isolated item on a data line, or in the body of certain directives such as `DO` and `IF`. These labels are actually removed in the "procedure compilation" process, so that CLIP in fact never sees them when reading a procedure. Details on this rather advanced topic are given in §6 of Volume II.

§6.9 DOLLAR SIGNS

The dollar sign is used as a special character in one instance: as prefix of the argument counter in the "definition body" of macrosymbols that admit arguments (§4 of Volume II). This function was previously performed by the percent sign.

REMARK 6.7

In previous CLIP versions the dollar sign served as a special character in two instances.

1. As prefix for registers, which were special integer items identified as $1, $2, ... $8.

2. When prefixed and followed by a blank, it acted as an end-of-data-field terminator (§5.3), and also as a comment line marker when used as sentinel (§4.1).

§6.10 EQUALS SIGNS

Equals signs, like blanks and commas, are item terminators but serve to specify *assignments*, as discussed in §3.1. They also terminate lists. The following two examples illustrate typical uses.

Example 1:

```
SET TIME = 0.2407
```

Here SET is an assignment command. The equals sign separates the destination item TIME from the source item 0.2407.

Example 2:

```
SET INITIAL /TIME=6.7 /HEIGHT= -0.34
```

Here the equals signs are used in the parameterization of qualifiers TIME and HEIGHT.

REMARK 6.8

In pre-1983 CLIP versions equals signs were treated exactly as blanks. Not so now; see next Remark.

REMARK 6.9

Here is the parsing of the SET command of Example 1:

Type	*Pre*	*Value*	*Sep*
Character		SET	
Character		TIME	=
Floating		0.2407	

It is seen that the equals sign is "remembered" as a separator character, which may be retrieved through the CCLSEP function described in Volume III.

§6.11 PARENTHESES

Balancing left-right parentheses serve four special purposes:

1. Argument list delimiters in PROCEDURE and CALL directives (Volume II). Example:

   ```
   *CALL SOLVER (A=2/3; FILE=START(3))
   ```

   Note that the parenthesis pair surrounding 3 is not a delimiter, because it does not balance the opening parenthesis; thus the item that follows FILE= is parsed as START(3).

2. Argument list delimiters in references to macrosymbols that accept arguments. For example:

   ```
   <ifdef(range; <exp(2)> ; range)>
   ```

3. Grouping in arithmetic expressions (§7.4), where they are also used to control the evaluation sequence.

4. Double list generators (§8.4)

Outside of these four cases, parentheses are treated like an ordinary nonnumeric character. For example:

```
PRINT OUTPUT*DAT(4)
```

OUTPUT*DAT(4) (a legal file name on some archaic computers) is interpreted as a single character string.

§6.12 PERCENT SIGNS

Percent signs have only one special function as integer-divide arithmetic operator (see §7.4). Aside from this case, percent signs are treated like any ordinary nonnumeric character.

§6.13 PERIODS

The period (also called dot) has only one special function. An isolated period is interpreted as end-of-data-field terminator (§5.3), and a period sentinel followed by a blank flags a comment line (§4.1).

Aside from this case, a period is used as an ordinary character that can appear in both numeric and nonnumeric items and expressions of all kinds.

§6.14 QUOTES

Quotes are used to delimit *quote strings*. Quote strings are used to implement *inline prompting* as explained in §8.5. Example:

```
OPEN "Enter filename: "
```

The quote string **Enter filename:** will appear on the screen as a prompt. Whatever you type in response to the prompt will replace the quote string. Thus if you respond **INPUT.DAT** then

```
OPEN INPUT.DAT
```

will be the actual command processed by CLIP. This technique is often attractive when scripts and/or command procedures are combined with interactive usage. It makes no sense in batch mode.

Quotes have a higher precedence than any other character except the double minus hyphen in the cases discussed in §5.3 and a *vis a vis* relationship with the apostrophe: Quotes inside an apostrophe strings are treated like ordinary characters, but apostrophes inside a quote string are treated as ordinary characters.

REMARK 6.10

On VAX/VMS you should beware of the following construction, which specifies a file name across a network:

```
user"name password"::disk:[directory]filename
```

The entire pathname should be enclosed in apostrophes to make it one string (note the blank after **name**) and to defuse the quotes.

§6.15 SEMICOLONS

Semicolons have two special uses:

1. Default argument delimiter in procedure and macro argument lists; for example

```
SET TIME = <max(<t>; 25.4)>
```

2. If prefixed by a blank, and not inside an apostrophe or quote string, it separates commands written on the same data line (§5.3).

Outside of these two cases, semicolons are treated like ordinary nonnumeric characters. For example,

```
PRINT OUTPUT.DAT;4
```

`OUTPUT.DAT;4` (a legal VAX/VMS file name) is interpreted as an ordinary character string.

§6.16 SLASHES

Slashes have two important special uses:

1. **Qualifier prefix in ordinary commands and directives.**
2. **Floating division operator in arithmetic expressions.**

Currently (see Remark below), slashes are item delimiters only after character strings. Thus, the expression

```
*DO/8
```

represents two items: the character string *DO and the integer qualifier 8. Another example:

```
OPEN/NEW/NOMINAL
```

represents three items: the verb OPEN and the qualifiers NEW and NOMINAL.

Slashes are not delimiters in expressions such as

```
(3/8)        (1/(2/5))
```

which represent two floating-point items whose value is 0.375 and 2.5, respectively. Details are given in §8.4.

§6.17 SQUARE BRACKETS

Balancing left-right square brackets have only one special use: delimiters that indicate formal-argument substitution in the body of a command procedure, as explained in §5 of Volume II. Example:

```
*PROCEDURE SOLVE (A;B;X)
 FACTOR [A]
 SOLVE [A] [B] = [X]
*END SOLVE
```

In this example, [A], [B] and [X] are to be replaced by actual argument text when procedure **SOLVE** is called.

Outside of this rather special case, square brackets are treated as ordinary nonnumeric characters. For example:

```
TYPE  DRDO:[FELIPPA.CLIP]TCL.TES
```

The item following **TYPE** is interpreted as a single character string, which the reader will recognize as a legal VAX/VMS file name.

7

Data Items

§7.1 CLASSIFICATION

A ***data item*** is a sequence of characters that represents a *single* and *constant* value. Each data item that appears in a command record is categorized by CLIP into one of three types: integer constant, floating-point constant, or character string.

The first two types are numeric and may be freely converted into each other when the processor calls for a numeric value. The last type is nonnumeric and may not be converted to numeric.

The processing of symbolic items such as macrosymbols and of certain special items such as list generators eventually reduces to the evaluation of one or more data items. This is true regardless of the complexity of the intermediate expressions.

This Section explains the formatting rules for data items. Special items are covered in §8 whereas symbolic items are discussed in Volume II.

§7.2 INTEGER CONSTANTS

An integer constant consists of a sequence of digits (0 through 9) possibly preceded by + or -. Examples:

365 -35767 +174

REMARK 7.1

Integers must be restricted to the legal range allowed by the host computer hardware. This range is typically -2^{n-1} to 2^{n-1}, where n is the number of bits in a FORTRAN integer word.

REMARK 7.2

Within CLIP, integers are stored as double-precision floating-point numbers on 32-bit machines and as single-precision floating-point numbers on 64-bit machines.

REMARK 7.3

The octal integer (recognized by a leading zero in ancient versions of CLIP) has disappeared.

§7.3 FLOATING-POINT CONSTANTS

Single-Precision

A single-precision floating-point constant consists of an integer part, a decimal point, a fraction part, an E, and an optionally signed integer exponent. The integer and fraction part both consist of a sequence of digits. Either the integer part or the fraction part, but not both, may be missing. Either the decimal point or the E and the exponent, but not both, may be missing. Examples:

```
16.07   32.   .0025   129.E+1 0123E+007   4.7E-23
```

"Borderline" items such as

```
11+2   -01-03   (both . and E missing)
```

are also interpreted as floating-point items 1100. and -0.001, respectively.

Double-Precision

A double-precision floating-point constant consists of an integer part, a decimal point, a fraction part, a D, and a signed integer exponent. The integer and fraction part both consist of a sequence of digits. Either the integer part or the fraction part, but not both, may be missing. The exponent mark D is mandatory. Examples:

```
16.07D0    123D+007   23D-6   +.47D-20
```

REMARK 7.4

CLIP stores *all* numeric items (integer, single floating and double floating) in floating-point form in its internal tables. The internal floating-point precision is double on all 32-bit machines and single on all 64-bit machines.

§7.4 ARITHMETIC EXPRESSIONS

Definition

An arithmetic expression is a data item of the form

$$(c_1 \odot c_2 \ldots \odot c_k)$$

in which c_1 through c_k are integer or floating-point constants (or symbolic expressions that eventually evaluate to such) and $\odot$ denotes one of the following one-character operators:

Character	*Operator*
+	addition
-	subtraction
*	multiplication
/	floating division
%	integer division
^	exponentiation

Examples:

```
(-2*3)    (1/5 3)    (4^.5)    (-4./24+1.20^2)
```

In the absence of internal parentheses, the indicated operations are performed *according to the hierarchical rules of FORTRAN*. That is, the operator hierarchy is: exponentiation (highest), multiplication/division, addition/subtraction (lowest).

```
(1+1/3)
```

evaluates to 4/3 = 1.333333 ... Internal parentheses may be used to override the operator hierarchy. For example,

```
((1+1)/3)
```

evaluates to 2/3 = 0.666666 ... .

Any of the c_i may be a symbolic item (macro symbol, register, procedure parameter) that eventually evaluates to a numeric value.

Result Typing

The type of the final result is either integer or floating-point. If *all* component values are integer *and* the operator / does not appear, the result is integer. Otherwise the result is floating-point.

Note that there are *two* division operators: / and %. The slash forces floating-point division and makes the result floating-point even if dividend and divisor are both integers.

The percent sign forces integer division as in FORTRAN. Consider for example the two items

(17/4)   (17%4)

The first item evaluates to floating 4.25 whereas the second one evaluates to integer 4.

If % is used on floating-point operands, both dividend and divisors are converted to integer before the division takes place and the result is typed integer.

REMARK 7.5

In versions of CLIP endowed with the macrosymbol facility, the macro-evaluation delimiter symbols < and > may be used instead of (and), respectively. The effect on item parsing and evaluation is identical. They are not equivalent, however, when "virgin" command lines are retrieved through the CLGET entry point as discussed in Volume III. A detailed explanation is given in §4.10 of Volume II.

REMARK 7.6

The exponent following ^ must be an integer if the base is negative. Thus -2.0^5. is illegal, but -2.0^5 is legal and evaluates to -32.0.

REMARK 7.7

An attempt to divide by exact zero will produce an error diagnostic and the division will be skipped.

REMARK 7.8

If the result is floating point, the arithmetic work is carried out in full double precision arithmetic and the result is stored in double precision on 32-bit machines while all arithmetic is done in single precision on 64-bit machines.

REMARK 7.9

Blanks encountered inside a parenthetical expression are ignored. For example

(2 * 8)

evaluates to 16.0; the blanks before and after * being ignored. But

2. * 8

does *not* evaluate to 16.0, and is in fact treated as three items: floating constant 2.0, character *, and integer 8.

REMARK 7.10

The following differences with previous versions of CLIP should be noted:

1. The exponentiation operator is now ^ instead of **.

2. Parentheses were not allowed in pre-1983 CLIP versions to control expression evaluation order.

3. The result was always of floating-point type.

§7.5 ORDINARY CHARACTER STRINGS

A data item that does not qualify as a numeric value is classified as a *character string*, or *string* for short. Ordinary character strings are those not surrounded by apostrophe or quote delimiters.

The following items are interpreted as ordinary character strings:

1. Any item that contains a nonnumeric character and does not qualify as a symbolic item or an arithmetic expression. Examples:

```
NODE  D66  STIFFNESS.FILE  Help  4+R  $$$  (1/E4)
```

2. A data item that contains only numeric characters but is not a valid integer or floating constant. Examples:

```
E6  -E6  2.3.4  2E3E4  8D.7  8D+.5
```

REMARK 7.11

All lower case characters present in an ordinary character string are converted to upper case as they are processed on the VAX/VMS version; they are not converted on the UNIX versions.

§7.6 APOSTROPHE STRINGS

A sequence of one or more characters surrounded by apostrophes, as in

```
'ABC'   '123'   ' '   'A rather long string'
```

is called an ***apostrophe string.*** Any graphic character enclosed between the apostrophe delimiters, with one exception, is interpreted as part of the string. The only exception pertains to the double-minus hyphenator (see §5.3).

To represent an internal apostrophe, repeat it as in FORTRAN 77. For example

```
'Don't get me wrong'
```

represents the string `Don't get me wrong`.

Lower case characters inside an apostrophe string are not converted to upper case.

§7.7 QUOTE STRINGS

A sequence of one or more characters surrounded by quote marks, as in

```
"Please say something ..."
```

is called a ***quote string***. Quote strings are used to implement *inline prompting*. This is best illustrated by an example. Suppose the following command is present in a script file or a command procedure:

```
OPEN  "File to open:  "  /"OLD, NEW or SCR:"
```

When this command comes up the two quote strings will appear on your screen as prompts, and CLIP will wait for your response:

File to open :	INPUT.DAT
OLD, NEW or SCR:	NEW

where the text on the right of the : are your assumed responses. The quote strings are replaced by your responses, so the command that CLIP will actually process is

```
OPEN  INPUT.DAT  /NEW
```

REMARK 7.12

Lower case characters inside a quote string are preserved in the prompt message. Double minuses are interpreted as ordinary characters if a closing quote appears on the same line; otherwise it is treated as a hyphenator. Apostrophes are treated as ordinary characters.

REMARK 7.13

Obviously the use of quote strings makes little sense in purely interactive work except as a plaything. Its main value is the "filling of blanks" in command procedures or script files. One especially useful application is in self-documenting procedure call sequences, as illustrated by

```
*CALL INTEGRATOR (  TBEG = "Starting time:"; --
                    TEND = "Target time:" ; --
                    DT = "Time increment")
```

§7.8 BORDERLINE CASES

This final subsection deals with the fringe elements. Some items are not easy to classify because they are in the grey zone between numeric and nonnumeric. For example:

```
+        .+24        (3/)
```

A general rule holds for these cases: *when in doubt, assume a character string.* Following are some consequences of this rule.

1. ***Isolated characters.*** Any isolated character that is not a digit (0 through 9) is classified as a character string. For example, the isolated operators

   ```
   +    -    /    :
   ```

   This interpretation simplifies the processing of algebraic language statements.

2. ***Impossible expressions.*** Constructions such as

   ```
   ( 4/)
   ```

   are interpreted as character strings.

8

Special Items

There are two types of special items:

1. ***Record marks***, which are used to specify continuation or explicit ending of command records.

2. ***Item generators***, which can be used as work-saving aids to replicate items or to generate regular sequences of numeric items.

§8.1 RECORD MARKS

The following are character sequences that function as record marks:

Mark	*Sequence*	*Function*
.	blank-period-blank	End of record; following text is ignored
++	blank-plus-plus	Continue record with item break
--	minus-minus	Continue record without item break
;	blank-semicolon	Separate records on same data line

For a more detailed description of these marks, see §5.3.

§8.2 ITEM REPETITION

A data item prefixed by *n*@, where *n* is an unsigned nonzero integer, is equivalent to repeating the item *n* times. Example:

```
4@64        2@DUM        3@(1/2)
```

This is the same as writing

```
64, 64, 64, 64    DUM, DUM    0.5, 0.5, 0.5
```

As the example indicates, the generated item sequence is interpreted as a comma-linked list. It can therefore be processed by one of the list-loading entry points described in Volume III.

REMARK 8.1

The item following *n*@ may be a symbolic item that eventually evaluates to an individual value. The count *n* can also be a symbolic item that eventually evaluates to a positive nonzero integer value.

REMARK 8.2

The item following *n*@ may not be another special item. For example, `6@5@2.5` will thoroughly baffle CLIP.

§8.3 SINGLE LIST GENERATION

List Generators

While preparing input data to application programs, there frequently arises the need for specifying lists of numeric items whose values are arranged in arithmetic or geometric progression. For example:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

6, 2, -2, -6, -10, -14

8.0, 4.0, 2.0, 1.0, 0.5, 0.25
```

If the list is fairly long, the use of item list generators can result not only in labor saving but, more importantly, in reducing the risk of key-in errors through the proven principle "let the machine do it."

Item generation of this *single* kind can be specified with a three-item construction in which two numeric data items called the *end values* are followed by a special item called the *step generator*. The following constructions are permitted:

Stepped arithmetic progression	v_1:v_2:s
Subdivision into equal intervals	v_1:v_2:/m
Geometric progression	v_1:v_2:*m

Here v_1 and v_2 are two numeric items of matching data type (integer or floating), which together define the first and last value of the generated list, respectively; s is an optionally signed numeric value of the same type as v_1 and v_2, and m is an unsigned nonzero integer. Any of these components may be a symbolic item that evaluates to a numeric value.

Arithmetic List: Explicit Step

The form v_1:v_2:s generates the arithmetic progression

$$v_1,\ v_1+s,\ \ldots\ v_1+ks, \quad \text{where} \quad v_1+ks \le v_2 < v_1+(k+1)s \quad \text{if} \quad s>0$$
$$v_1,\ v_1+s,\ \ldots\ v_1+ks, \quad \text{where} \quad v_1+ks \ge v_2 > v_1+(k+1)s \quad \text{if} \quad s<0$$

Using this construction, the first two example lists above can be abbreviated to

```
1:12:1        6:-14:-4
```

The general form of this generator is perhaps easier to remember by thinking of the FORTRAN DO loop

DO *label* 1,12,1   DO *label* 6,-14,-4

If the step s is omitted, a unit increment is assumed. Thus 1:12:1 may be further shortened to 1:12.

Arithmetic List: Subdivision into Equal Intervals

The form v_1:v_2:/m generates an arithmetic progression by subdividing the interval $v_2 - v_1$ into m parts:

$$v_1,\ v_1 + (v_2 - v_1)/m,\ v_1 + 2(v_2 - v_1)/m,\ \ldots\ v_2$$

If $m = 1$ (or $m \leq 0$) no generation occurs and the list reduces to the end values.

Using this form the first two example strings may be written

```
1:12:/11        6:-14:/5
```

This form is generally preferable to the v_1:v_2:s form if the items are of floating-point type and the number of subdivisions is more easily visualized than the step value. For example, typing

```
1.0:64.0:/25
```

is less error prone than saying

```
1.0:64.0:2.52
```

because rounding errors may cause the last generated item to miss the 64.0 target. Another advantage is that the user need not be concerned as to whether the resulting step is positive or negative.

Geometric List Generation

The form v_1:v_2:*m generates a geometric progression going from v_1 to v_2 with the ratio $(v_2 - v_1)^{1/m}$. The net effect is that $m - 1$ values are inserted, and the interval $v_2 - v_1$ subdivided into m logarithmically identical intervals. For example, the third example list in §8.3 may be generated by writing

```
8.0:0.25:*5
```

In this form, both v_1 and v_2 should be nonzero and have the same sign: they are always interpreted as floating point numbers. If $m = 1$, the list reduces to the end values.

§8.4 DOUBLE LIST GENERATION

The single list generation capability described in §8.3 is equivalent to a one-level DO construction. The double list generation capability described herein is equivalent to a two-level (nested) DO construction. This form does not appear as frequently in practice as single list generation, but it's handy to have around should the need arise.

Double list generation is best explained by an example. Consider the 10-integer list

```
3,8, 5,7, 7,6, 9,5, 11,4
```

This is composed of two interlaced arithmetic progressions: `3:11:/4` and `8:4:/4`. But if one tries the abbreviation

```
3:11:/4, 8:4:/4
```

the result is not what you want:

```
3,5,7,9,11, 8,7,6,5,4
```

The interlaced list can be generated by the construction

```
(3,8):(11,4):/4
```

The general form is

$$(list_1):(list_2):/m$$

where the following restrictions apply:

1. $list_1$ and $list_2$ are numeric lists that contain the same number of numeric items (up to 16). These items may be specified explicitly or through repeat-item or single-list-generation constructions.
2. m is an unsigned nonzero integer that specifies the number ($m-1$) of intermediate sublists to be generated. If $m = 1$ the generated list reduces to the end values. This item *must* be specified; no default is accepted.

Blanks that occur inside the delimiting braces are ignored.

The following examples illustrate how this construction works.

`(1,2,3):(13,-7,15):/3`	`1,2,3, 5,-1,7, 9,-4,11, 13,-7,15`
`(3@1):(1:11:5):/2`	`1,1,1, 1,3,6, 1,6,11`
`((3/2),0):(-(1/2),0):/4`	`1.5,0, 1.0,0, 0.5,0, 0,0, -0.5,0`

9

Lists

§9.1 WHAT IS A LIST?

The concept of *item list*, or simply *list*, is important for many CLIP-supported processors. Conceptually a list is a sequence of items bearing a "connection" relationship. This relationship is established in two different ways.

If the items are explicitly typed one by one, the list attribute is conferred by separating them with commas. If the items are generated through the work-saving constructions described in §8.2–8.4, the list attribute is conferred implicitly.

There are two types of list: numeric lists and character lists, which are described in the following subsections.

§9.2 NUMERIC LISTS

The usual way of specifying a numeric list is through the comma connective. Example:

```
1,2,4,8,16,32
```

This is a list of six integers. Since CLIP keeps internally all numeric data in floating-point format, floating-point numbers and integers can be freely mixed in a numeric list. Thus

```
1.0, 2.0, 0, 36.0E0, 6/2
```

is the same as

```
1, 2, 0.00, 36, 3
```

Blanks that appear before and after the comma connective are ignored. Consecutive commas, or commas separated only by blanks, generate *zero items*. For example, the following list

```
1,, 2, ,,6
```

is the same as

```
1,0,2,0,0,6
```

Numeric and character-string items cannot *normally* be mixed in the same list. For example:

```
1,2, ABC, DEF, 6, 7
```

For normal item loading, there are actually three lists here:

```
1,2
ABC, DEF
6,7
```

The commas after 2 and DEF are irrelevant. But in certain contexts mixed lists are acceptable; in fact this happens in many of the directives discussed in Volume II.

The numeric item generators described in §8.3–8.4 also generate lists. For example:

```
0.4, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 8.0, 8.0, 8.0
```

can be abbreviated to

```
0.4, 1.:6., 3@8
```

inasmuch as the generated items are assumed to be connected by commas.

REMARK 9.1

A numeric list may be loaded by the processor in one of three modes: integer, single-precision floating, or double-precision floating. CLIP delivers the goods in the requested format after transforming them as appropriate. Note that it is always the *processor* that decides on which data type is best for its own consumption; CLIP simply supplies the data in the form it is told to.

§9.3 CHARACTER LISTS

Character lists occur less often than numeric lists in practice. They are constructed with the comma connective. For example

```
AA, BB, CC, DD, EE
```

is a five-item character list.

In previous CLIP versions, the slash connective was also considered as producing a character list. Thus

```
AA/BB/CC/DD/EE
```

was identical in all respects to **AA,BB,CC,DD,EE**. However in the present CLIP version, only the comma-connected form is treated as a character list. The slash-connected form is treated as character string **AA** followed by four character qualifiers: **BB**, **CC**, **DD** and **EE**.

REMARK 9.2

To make the distinction more precise, here is the parsing of **AA,BB,CC,DD,EE** as stored in the Decoded Item Table:

Type	*Pre*	*Value*	*Sep*
Character		AA	,
Character		BB	,
Character		CC	,
Character		DD	,
Character		EE	

and here is the parsing of **AA/BB/CC/DD/EE**:

Type	*Pre*	*Value*	*Sep*
Character	/	AA	
Character	/	BB	
Character	/	CC	
Character	/	DD	
Character	/	EE	

10

Command Description

§10.1 MOTIVATION

The developer of an application program (for example, a NICE Processor) that uses CLAMP as source input language needs a *metalanguage* for describing legal commands when a user manual is prepared or an online help file is written. (A metalanguage is a language used to define another language; natural languages, such as English, are in fact metalanguages.)

The CLAMP metalanguage was originally patterned after the COBOL metalanguage, particularly with regard to the use of special symbols such as square brackets and braces. Since then, the metalanguage evolved into a simpler and more natural form as a result of experience in preparing NICE help files. Only a few of the original rules have survived.

This Section presents rules for the description of CLAMP commands in user's manuals or help files. These rules also apply to the description of directives, which are covered in Volume II.

§10.2 DESCRIPTION SYMBOLS

Symbols that represent individual data items or item lists are written as alphanumeric strings. The use of upper or lower case letters in these strings *is* meaningful.

Keywords

Upper case words are mandatory keywords that must be entered as indicated. For example:

```
RESTART
```

Keywords can be frequently abbreviated to a non-unique "root". The convention generally followed within the NICE system is that if more characters than the root are given, they must agree with the full spelling. For a full discussion of the subject, see §4 of Volume III.

Unlike previous metalanguage versions, the root is no longer shown explicitly in the command description. Such refinement was found unnecessary for an interactive system in which the user can quickly find out about the root by experimentation.

Variable Character Strings

Capitalized lower case words that do not contain a "list" represent a variable character string. Example:

DEFINE TITLE = *Text*

Here *Text* represents the expected text of a title input, which can of course be virtually anything. In many cases restrictions are placed upon which strings are acceptable. For example:

SET FREEDOM = *Idof*

The symbol *Idof* (freedom identifier in a finite element program) may be TX, TY, ... etc. Such restriction must of course be described in the help file or user's manual.

Numeric Items

Lower case words that do not contain a "list" suffix represent a single numeric value. Example:

DELETE NODE *n*

FORTRAN typing conventions are often followed to distinguish integer values from floating-point values should the distinction be important (in many cases it is not). This is a matter of personal style that is not subject to metalanguage rules.

In typeset text (such as this Manual), numeric item symbols are expressed in *italics*. This convention makes references to *n* in the text more vivid. Of course, this is not possible in computer-stored help text, where one has only typewriter fonts available.

Numeric Lists

There are two ways of specifying numeric lists. One is to use a single lower case word that has a "list" suffix. Example:

DEF NODE n = *coordinate-list*

Another is to show the list explicitly:

DEF NODE $n = x_1, x_2, x_3$

This works well for short, fixed-size lists in typeset Manuals (like this one) since subscripts look classy. For variable length lists, one may use the "three-dot" notation:

DEF NODE $n = x_1, x_2, \ldots x_k$

Again this looks best on a typeset page, but is not so good on a computer-stored text file.

Character Lists

This is similar to the numeric list case. One way is to use a capitalized lower case word that has a "list" suffix. Example:

PRINT [*Optionlist*]

Another is to show the list explicitly:

PRINT *Node-switch, Element-switch, Freedom-switch*

DELETE $Component_1, \ldots\ Component_k$

Framing

Metalanguage statements are often framed to attract attention. Like this:

DELETE *nodelist*

§10.3 METASYMBOLS

Optional Items

Square brackets are used to indicate that the intervening expression(s) are optional. Example:

PRINT [*Optionlist*] *Filename*

If omitted, appropriate defaults are assumed by the user program. In the standard CLAMP format described in §3.1, command *qualifiers* are always optional.

Mutually exclusive options, which may not coexist in the same command, are separated by vertical bars:

CONNECT DEVICE [BLOCK-IO | FORTRAN-IO] = *unit*

Mandatory Choices

Curly braces indicate that a choice from the enclosed expressions (separated by vertical bars) must be made. Example:

SET MOOD = {TAKEITSERIOUSLY | TAKEITEASY}

Control Characters

Control characters = (assignation) and / (qualifier prefix) are shown in the command description statement. Similarly, commas must be show if item lists are explicitly specified.

A

Glossary

§A.1 GLOSSARY

The terms defined below include those used in the present Volume as well as some, like *database* and *script*, which are mentioned in passing in this Volume but figure more prominently in Volumes II and III.

apostrophe string — A character string enclosed between apostrophe marks. All characters inside an apostrophe string (with the only exception of hyphenation marks) are significant. Lower case letters are protected.

architecture — For a software product, the specification of the user interface. F. P. Brooks in his classical *The Mythical Man-Month*, defines architecture of a system as "... *the complete and detailed specification of the user interface. For a computer this is the programming manual. For a compiler it is the language manual. For a control program it is the manuals for the language or languages used to invoke its functions. For the entire system it is the union of the manuals the user must consult to do his entire job.*"

argument — See *macrosymbol*, *procedure argument*.

arithmetic expression — A sequence of numeric constants enclosed in parentheses and separated by arithmetic operators. Internal parentheses may be used to specify subexpressions and force certain evaluation sequences. An arithmetic expression evaluates to an integer constant or a floating-point constant according to the rules stated in §7.4.

arithmetic operator — One of the symbols +, -, *, ^, % and /, which are used to specify operations in arithmetic expressions.

assignment command — A command that specifies a value-assignment action. In the standard CLAMP format, this is expressed by connecting two parameters (or parameter lists) by an equals sign.

batch run — A run mode in which a processor is under exclusive control of an operating system scheduler. Contrast to *interactive mode*.

character — The symbols that comprise an alphabet: letters, numbers (also called digits or numerals), and marks. On the computer, the concept is extended to include control (nongraphic) symbols encoded according to a standardized scheme.

character string — A data item interpreted as a sequence of characters.

CLAMP — Acronym for **Command Language for Applied Mechanics Processors**.

CLIP — Acronym for **Command Language Interface Program**. The component of the NICE architecture that implements CLAMP.

CLIP operation mode — The plan of action followed by the CLIP kernel in response to the type of command being processed (ordinary command or directive) and its sender (user or processor). Operational details are given in Volume III.

closed interface A software system interface that forbids global variables.

column The index of a data line character, counting from left to right and starting at 1. (Terminology holdover from the days of punched card input.)

command In a command language: an instruction consisting of one or more items to be interpreted by the program that receives it.

command language An interpretable language consisting of a stream of commands that controls the execution of a software element.

command record The logical representation of a command as a set of items. In CLAMP, a finite sequence of items terminated by an implicit or explicit end of record. A command record must contain at least one item.

command source file The input file from which CLIP reads data lines sent by the user or the processor.

command source stack A stack structure used to implement multiple source input as explained in §4.2.

command stream The sequence of command records "seen" by CLIP, when abstraction is made of physical input source.

comment line A data line flagged by a comment sentinel.

comment sentinel A mark in column 1 of a data line that identifies the text that follows as comment. In CLAMP, the default mark is the period when followed by a blank or a carriage return.

comment text Text present in the command stream which is ignored by CLIP. The following are treated as comments: text outside data field; text that follows a continuation or end of line mark; text on a data line that contains a comment sentinel; text that follow certain "one liner" directives described in Volume II.

continuation mark One of the special items ++ or --, which specifies that the current command record continues on the next data line. The double plus mark must be preceded by a blank and always breaks items; the double minus mark is a hyphenator and does not break items.

conversational run A form of interactive work in which a human user maintains a dialog with a running program.

data The representation of information on a digital computer as stored values.

database A named collection of stored data organized according to a data model.

data library A named partition of a database, which can be attached to a running processor as an entity. A data library normally resides on a permanent file.

data line Each physical record read by CLIP from the command source file. These records do not normally exceed 80 characters under default settings.

data field	The active portion of a data line.
data item	An item that is directly translated into a numeric or character-string value.
data manager	A software element that stores, retrieves or maintains data structures. If the structures form a database, the data manager is called a database manager.
directive	A special command record that is directly processed by CLIP and not transmitted to the running processor.
empty line	A data line that contains only blanks or comment text.
end of line	A special item which terminates a command record and indicates that the next one begins on another line. In CLAMP, the default end-of-line mark is the isolated period.
end of record	Any character or character sequence that indicates the end of a command record. The end of the record may be explicitly written with a special item (for example, end of line or record separator), signalled by a carriage return mark in terminal input, or implicitly given by the end of the data field being reached without a continuation mark having been detected.
end of source	Any signal that marks the termination of the current command input source. Examples: an end of file in a script, a **RETURN** directive in a command procedure, a data line containing "@" in column 1.
fixed-point constant	See *integer*.
floating-point constant	A data item that is identified and decoded as a floating-point value. A floating-point constant may be written in the usual FORTRAN style, or be the result of an arithmetic expression.
GAL	Acronym for **Global Access Library**, which is a data library that conforms to the data model of the NICE global database. Also, the name of the database manager through which GAL files are accessed.
global database	A database that resides on permanent storage and is accessible by a network of processors.
global data manager	A data manager through which the global database is accessed.
hyphen	See *continuation mark*.
input file	See *command source file*.
interpreter	A software element that translates a source language into a target language on a record-by-record basis under the supervision of an external control structure.
integer	A data item that is interpreted as a fixed-point value.
interactive run	A run mode in which the processor is under direct control of a human user. This can be further classified into *conversational run* and *spectator* or *monitor run* according to the degree of interaction.

item
: A finite sequence of characters parsed as a token. In CLAMP, items other than apostrophe strings, quote strings, procedure arguments or macrosymbols are delimited by blanks, commas, equals signs, qualifier prefixes, list-generation prefixes, end-of-record marks or data field boundaries. Apostrophe and quote strings are delimited by a matching apostrophe, a matching quote, or the end of the data field. Arguments and macrosymbols are delimited as explained in Volume II.

item list
: A sequence of items separated by commas.

log file
: A file on which CLIP writes a transcript of the commands it reads.

kernel
: In a NICE Processor, the software that performs the useful work. The kernel is surrounded by the ***shell***, which interfaces it to the architecture. (Terminology suggested by the Unix system.)

keyword
: A character string that triggers a specific action or response from the command interpreter on account of its spelling.

line
: See *data line*.

list
: See *item list*.

list generator
: A special-item construction such as 1:15:2 that evaluates to a numeric list the items which are an arithmetic or geometric progression. Generation may be one-dimensional (single list generation) or two-dimensional (double list generation).

macrosymbol
: A character string that stands for another character string. The replacement process is called macro expansion and may involve argument-passing and recursion. This process is explained in Volume II.

mailbox mode
: An advanced variant of the message mode in which the running processor uses CLIP as a "mailbox" to send commands to another processor.

message mode
: An operating mode in which the processor "talks" to CLIP by calling a "message" entry point.

metalanguage
: A language used to define another language. The CLAMP metalanguage is used to describe command records processable by CLIP.

metasymbols
: Special characters used in the metalanguage to specify logical properties but which are not part of the command as written. For example, square brackets are metasymbols used to indicate that the intervening expression(s) are optional.

monitor run
: See *spectator run*.

multiline record
: A command record that extends over more than a data line.

numerals
: Characters 0 through 9.

numeric character
: A character that may legally appear in integer or floating-point constants: numerals 0 through 9, +, -, ., E or D (E and D may not be the first character).

open interface	A software system interface that admits global variables such as FORTRAN common blocks.
operator	See ***arithmetic operator***.
ordinary command	A command that is not a directive. An ordinary command is not conceptually interpreted by CLIP, but passed along to the processor.
parameter	In a command language, a data item whose value is not specified in the command description, but is assigned when the command is written.
problem-oriented language	A command language that directs the activity of an application program or a network of such program, and that consists of domain-specific statements.
processor	A software element that receives and produces data structures. In the NICE system, a Processor (capitalized) is a software element that produces results for the user and conforms to certain operational rules.
processor directive	A directive submitted by the processor as a message.
procedure	A set of command records delimited by a procedure header and terminator and which may be parameterized by arguments specified in a calling sequence.
procedure arguments	A list of parameters specified in the procedure header and that may be used to control a call-by-name text-replacement mechanism.
procedure body	The set of commands comprised between the procedure header and terminator.
procedure header	The directive that initiates the definition of a command procedure.
prompt	In conversational operation of a processor, text that CLIP writes to the screen to indicate that it is ready to accept a command.
qualifier	An item (normally a character string) preceded by a qualifier prefix, which is by default the slash. Qualifiers are used to implement command options.
repeated item	An item of the form $n@item$, where n is a positive integer and $item$ a valid data item. This is equivalent to a list of n identical items. The at-sign is the default repetition character.
running processor	The processor that is under execution and calls CLIP for commands.
script	A file of commands that is prepared in advance and then inserted in the command stream by an **ADD** directive. A script differs from a procedure in that it cannot be parameterized or executed in nonsequential order.
sentinel character	A character that assumes a special role by appearing on column 1 of a data line.
shell	In a NICE Processor, the software that surrounds the kernel and communicates with the NICE architecture software. (Terminology suggested by the Unix system.)

software element Any piece of software that can be distinguished and identified for functional purposes. This may range from primitive subroutines to complex packages such as a database manager.

software system A software element or set of software elements packaged within a common architecture.

source See *command source file*.

special item A character sequence that does not evaluate directly to a numeric or character string value but is used for special purposes.

spectator run A form of interactive work in which the user does not actively interact with the running processor. Instead the user starts execution, designates input sources where the commands are prepared in advance, and "sits back" to watch the processor do its thing. Also called *monitor run*.

splash line An explanatory line (or lines) of text that is optionally printed by CLIP before the prompt to guide the user in command selection.

statement The conceptual representation of a command. More specifically, a command record or a set of command records when viewed as an element of a problem-oriented language. The view is in terms of actions in the domain of applications.

string A finite sequence of symbols that belongs to a common class. Also short for *character string*.

symbolic item An item that stands for another item or an item list.

text dataset A database entity that consists of a sequence of card-image records.

textstring A "passive" character string interpreted as data; for example a line of text or a plot title. Contrast to *keyword*, which is an "active" character string that controls program actions.

user The beneficiary (normally a human) of the processor activity.

user command An ordinary command submitted by the user.

user directive A directive submitted by the user; contrast to *processor directive*.

B

Ancient History

§B.1 HOW CLIP CAME UNTO BEING

An Illustrious Ancestor

The kernel of CLIP is LODREC. The initial version of LODREC was written by the author in 1969 while at the Stress Research Group of Boeing's Commercial Airplane Division (Seattle, Washington). The program was largely based on a punched-card free-field reader written by Lawrence Schmit, one of the architects of Boeing's ATLAS system.†

The first Univac version of LODREC was the result of converting the CDC version when the author moved to Lockheed's Palo Alto Research Laboratory in 1971. This version was documented in April 1971. Since then, successive versions of LODREC have been used as utility modules for processing the source input data of all of the application software written by the author. In fact, the author has not had the occasion of using a formatted READ for input data since 1969!

A major revision and expansion of LODREC took place during 1971-1972 while work on the now defunct NOSTRA (NOnlinear STRuctural Analyzer) program was underway. Many of the syntactical features which are now part of CLAMP took shape. It was decided to label the underlying language as NIL (NOSTRA Input Language), a designation that survived the NOSTRA code proper until 1978. A detailed documentation of NIL was published in 1973.

Another major revision of LODREC took place in 1976-1978. In 1976, the concept of *directive* was introduced as a way of implementing "service commands" intended for internal consumption by LODREC, and hence invisible at the user program level. The most important class of directives pertains to the definition and handling of *command procedures*, a concept implemented in late 1976. Further refinement of this feature occurred in 1978, when the ability for directly interfacing LODREC with a library-oriented database management system was established. The command-procedure concept proved to be so powerful that it led to the dropping of other experimental features (*e.g.* inline command generation), which are now more naturally presented in a procedure framework.

The Survival of the Fittest

During its nearly 10-year existence, LODREC has processed several million command records. New features were incorporated and tested almost every year. Only about half of those features have survived to date, as witnessed by the following list which covers LODREC and CLIP.

1. Multiple command per line (1969 to date)
2. Multiline commands (1969 to date)
3. Text records (1970-1974)
4. Parenthesized comments (1970-1972)
5. PL/1-like comments (1970-1972)
6. Starred character strings (1971-1974)
7. Packed-bit items (1971-1972)
8. Record generation by $*+k$ (1970-1976), superseded by 23.
9. Item generation by $*k=n+s$ (1971-1978)

† An evolved version of the first LODREC is still used as input data interpreter for ATLAS, which however runs only on CDC Cyber machines. Yet another derived version now drives the data management system RIM, developed by Boeing for NASA Langley.

Report Documentation Page

NASA
National Aeronautics and Space Administration

1. Report No. NASA CR-178384	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle The Computational Structural Mechanics Testbed Architecture Volume I - The Language		5. Report Date December 1988
		6. Performing Organization Code
7. Author(s) Carlos A. Felippa		8. Performing Organization Report No. LMSC-D878511
9. Performing Organization Name and Address Lockheed Missiles and Space Company, Inc. Research and Development Division 3251 Hanover Street Palo Alto, California 94304		10. Work Unit No. 505-63-01-10
		11. Contract or Grant No. NAS1-18444
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Current affiliation: Carlos A. Felippa, Center for Space Structures and Controls, Campus Box 429, University of Colorado, Boulder, CO 80309-0429

Langley Technical Monitor: W. Jefferson Stroud

16. Abstract

This is the first of a set of five volumes which describe the software architecture for the Computational Structural Mechanics Testbed. Derived from NICE, an integrated software system developed at Lockheed Palo Alto Research Laboratory, the architecture is composed of the command language (CLAMP), the command language interpreter (CLIP), and the data manager (GAL). Volumes I, II, and III (NASA CR's 178384, 178385, and 178386, respectively) describe CLAMP and CLIP and the CLIP-processor interface. Volumes IV and V (NASA CR's 178387 and 178388, respectively) describe GAL and its low-level I/O. CLAMP, an acronym for Command Language for Applied Mechanics Processors, is designed to control the flow of execution of processors written for NICE. Volume I presents the basic elements of the CLAMP language and is intended for all users.

17. Key Words (Suggested by Authors(s)) Structural analysis software Command language interface software Data management software	18. Distribution Statement Unclassified—Unlimited Subject Category 39

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	93	A05

NASA FORM 1626 OCT 86

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171

10. Item repetition by $n * item$, then $n@item$ (1970 to date)
11. Mandatory data line sentinels (1972 1978)
12. Composite floating-point constants (1972-1984), superseded by 29.
13. Apostrophe textstrings (1972-to date)
14. Numeric local variables (1972-1979), superseded by 25.
15. Local symbols (1973-1980), superseded by registers.
16. Transfer to indexed record (1974 1976), superseded by 21.
17. Record group repetition (1974-1976)
18. Hollerith textstrings (1974 1981)
19. Directives (1976-to date)
20. Command procedures (1976 to date)
21. Transfer to labels (1976-to date)
22. Colon-delimited item generators (1977 to date)
23. Record generation by `DO` directive (1977 1984), superseded by `GEN`.
24. Interface to global database manager (1978 to date)
25. Macrosymbol facility (1979 to date)
26. Registers (1979-1984), superseded by local macrosymbols.
27. Quote strings (1982-to date)
28. Structured directives `IF THEN ELSE`, `WHILE DO` (1982 to date).
29. Arithmetic expressions (1984 to date)

The acid test for survival of a new feature has been its usefulness and mnemonic quality in interactive work. If a person sitting at a terminal has to think for awhile before using a certain feature, doubts about its survival in the next version arise. Features found useful over several years may also disappear as a subsequent improvement is developed; for example, numeric local variables replaced by registers replaced by macrosymbols.

The transmutation of LODREC into CLIP took part in two stages. Functional requirements were identified as a result of the top-down *design* of the NICE architecture in the period March, 1979 through February, 1980. As the design evolved, it became evident that the command interpreter would have to be configured as a Unix-like "shell" surrounding the basic kernel (the old LODREC) as well as satellite subsystems for command-procedure handling, database management interface, etc. This ensemble was identified as CLIP.

The second stage involved the *implementation* of CLIP on the VAX 11/780 computer in the FORTRAN 77 language. The bulk of this work was carried out from March through August 1980. In retrospect, the decision of going with FORTRAN 77 (then just available on the VAX but not on Univac) was fortunate. The powerful FORTRAN 77 character-string processing capabilities allowed machine-independent coding of critical subroutines, and resulted in a productivity increase estimated at 3:1 over a similar effort that would had mixed FORTRAN 66 and assembly language. And over 90% of CLIP *is* character processing.